Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 216 (2008) 45–63
www.elsevier.com/locate/entcs

Inferring Static Non-monotone Size-aware Types Through Testing

Olha Shkaravska[1,2] and Marko van Eekelen[1,2]

Institute for Computing and Information Sciences
Radboud University Nijmegen

Abstract

We propose a size analysis procedure that combines testing and type checking to automatically obtain static output-on-input size dependencies for first-order functions. Attention is restricted to functions for which the size of the result is strictly polynomial, not necessarily monotone, in the sizes of the arguments.
To infer a size dependency, the procedure generates hypotheses for increasing degrees of polynomials. For each degree, a polynomial is defined by a finite number of points. Based on interpolation theory, in this paper we establish an upper bound on the number of test runs and a correct choice of test data that guarantees that all polynomials representing sizes of output lists can be found. The resulting hypothesis is then checked using an existing type checking procedure.
The procedure is not tied to the current size-aware type checker. The size-aware type of a function will be inferred if it exists and if it is accepted by a size-aware type checker. For terminating functions, our size-aware type inference procedure is complete with respect to type checking: if a function is well-typed, then the inference procedure terminates and produces corresponding size dependencies.

Keywords: Memory complexity analysis, type checking, testing

1 Introduction

Embedded systems or server applications often have limited resources available. Therefore, it can be important to know in advance how much time or memory a computation is going to take, for instance to determine how much memory should at least be put in a system to enable all desired operations.

Such decisions can only reliably be based on formally verified upper bounds of the resource consumption. However, an advanced detailed analysis of these bounds requires knowledge of the sizes of the data structures used throughout the program [6]. Trivially, the time it takes to iterate over a list depends on the size of that

[1] Email:R.vanKesteren@cs.ru.nl, O.Shkaravska@cs.ru.nl, M.vanEekelen@cs.ru.nl

[2] This research is sponsored by the Netherlands Organisation for Scientific Research (NWO), project Amortized Heap Space Usage Analysis (AHA), grantnr. 612.063.511.

doi:10.1016/j.entcs.2008.06.033

list. In this paper we focus on the task of automatically deriving the exact output-on-input size dependencies of function definitions in a program. The ratio behind exactness is explained later in this section. A possible relaxation of it is considered in Section 5.

Size dependencies can be represented in function *types*. We focus on shapely functions, where shapely means that the size relations are exactly polynomial (not necessarily monotone). The size of a list is its number of nodes (its length).

Consider examples. The function `progression` appends all tails of an argument list. Given the list `[1, 2, 3]` it returns the list `[3, 2, 3, 1, 2, 3]`. Thus the size of the output list is the sum of all integers from 0 to s (*arithmetic progression* $0 + 1 + ... + s$), where s is the size of an input. This explains the name of the function.

The function `cprod` computes the Cartesian product of two lists. It generates all pairs of elements, one taken from the first list, the other from the second. To define `cprod` one needs an auxiliary function `pairs`. The function `sqdiff` returns the Cartesian product of an argument with itself, if another argument is empty. If both arguments are not empty, then it recursively calls itself on their tails.

```
progression []       =  []
progression (x:xs)   =  progression xs ++ (x:xs)

pairs (x, [])        =  []
pairs (x, y:ys)      =  [x,y]:pairs (x, ys)

cprod ([], ys)       =  []
cprod (x:xs, ys)     =  pairs (x, ys) ++ cprod (xs, ys)

sqdiff (xs, [])      =  cprod (xs, xs)
sqdiff ([], ys)      =  cprod (ys, ys)
sqdiff (x:xs, y:ys)  =  sqdiff (xs, ys)
```

Given lists of size 3 and 2, for `cprod` the output is a list of size $3 * 2 = 6$ whose elements are pairs, i.e., lists of size 2, and the output for `sqdiff` is the list of size 1 of lists of size 2.

```
cprod ([1,2,3], [4,5])   =  [[1,4],[1,5],[2,4],[2,5],[3,4],[3,5]]
sqdiff ([1,2,3], [4,5])  =  [[3,3]]
```

The *size-aware type* of a function expresses the relation between its argument and result sizes. For instance, when two input lists have size s_1 and s_2 respectively, the output of `cprod` is a list of lists, with an outer list of size $s_1 * s_2$ and inner lists all of size 2. The output of `sqdiff` is a list of lists, with an outer list of size $(s_1 - s_2)^2$ and inner lists all of size 2:

$$\begin{array}{lcl}
\texttt{progression} & : & \texttt{[Int]}^{s} \to \texttt{[Int]}^{\frac{s*(s+1)}{2}} \\
\texttt{cprod} & : & \texttt{[Int]}^{s_1} \times \texttt{[Int]}^{s_2} \to \texttt{[[Int]}^{2}\texttt{]}^{s_1 * s_2} \\
\texttt{sqdiff} & : & \texttt{[Int]}^{s_1} \times \texttt{[Int]}^{s_2} \to \texttt{[[Int]}^{2}\texttt{]}^{(s_1 - s_2)^2}
\end{array}$$

In general, all lists at the input side, before the arrow, have an associated size variable. After the arrow, at the output side, all lists have an associated polynomial that determines the size of the output list. These polynomials are defined in terms

of the input size variables. The current presentation is limited to a language over lists for reasons of simplicity; size-aware types are straightforwardly generalized to general data structures and other programming languages.

Recently, we have developed a size-aware type checking algorithm to formally verify polynomially size-aware types (section 2) [12]. Given a size-aware type, the algorithm automatically checks if the function definition satisfies that type. Unfortunately, inferring such types is a lot more challenging than type checking and the type system approach does not straightforwardly extend (section 2.3). Therefore, we have suggested an alternative method of inferring size-aware types [12]. This paper develops this method into a practical type inference procedure.

The method is based on the observation that it is relatively easy to generate hypotheses for an *exact* size dependency by testing. Exactness (or strictness) of sizes makes it possible to place sizes of run-time tests exactly on the dependency graph. Because a polynomial of a given degree is determined by a finite number of values, its coefficients can be computed from the output sizes of run-time tests (figure 1). If the size expression is indeed a polynomial of that degree, it can be only *that* polynomial. This theory is used to create a practical procedure that yields hypotheses for size-aware types (section 3).

Combining hypothesis generation and type checking yields a procedure that can infer the size-aware type of a function (section 4). The procedure generates hypotheses for an increasing degree. For each degree, hypotheses for all polynomial size expressions in the output type are determined. The resulting size-aware type is checked using the size-aware type checking procedure. Thus:

(i) Infer the underlying type (without sizes) using standard type inference;

(ii) Annotate the underlying type with size variables;

(iii) Assume the degree of the polynomial;

(iv) For every output size annotation: determine which tests are needed, do the required series of test runs and compute the polynomial coefficients based on the test results;

(v) Annotate the type with the size expressions found;

(vi) Check the annotated type;

(vii) If checking fails, repeat from step 4 assuming a higher degree.

Indeed, for terminating programs the procedure is only guaranteed to find the

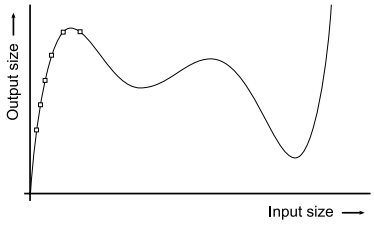

Fig. 1. A fifth degree polynomial is determined completely by any six of its points.

size-aware type if it exists. In practice, an upper limit on the degree can be used as a stopping criterion. Note that the procedure can also be applied with any other type checker for polynomially size-aware types.

The main contribution of this paper is developing the method suggested in [12] into a practical size-aware type inference procedure. Specifically, this means dealing with cases where the function definition only *partially defines* the output size polynomial: when the output type is a nested list and the output value is the empty list, there is no information on the sizes of the inner lists, like, for instance, in the case $\texttt{[[Int]}^?\texttt{]}^0$. We adopted the results from interpolation theory [3], on existence and uniqueness of polynomial interpolations, to define sets of test data that determine output polynomials in the right way.

2 Size-Aware type checking

Essentially, our approach to size-aware type inference for shapely functions is based on reducing inference to size-aware type checking. This section briefly describes the existing strict size-aware type system for a functional language and accompanying type checking procedure [12] that we use in the inference procedure. This also motivates our approach to type inference.

2.1 Size-Aware Types

The zero-order types we consider are integers, *strictly* sized lists of integers, *strictly* sized lists of *strictly* sized lists, etc. A strict list of length n is a list exactly of length n (not of some length up to n, as, e.g. in *sized* types of Pareto [11]). For lists of lists the element lists have to be of the same size and in fact it would be more precise to speak about matrix-like structures, e.g. the type $\texttt{[[Int]}^3\texttt{]}^2$ is given to a list which two elements are both lists of exactly three integers, such as `[[2,5,3], [7,1,6]]`.

$$\mathit{Types}\ \tau ::= \texttt{Int} \mid \alpha \mid \texttt{[}\tau\texttt{]}^p \qquad \alpha \in \mathit{TypeVar}$$

Here p denotes a *size expression*, i.e. a polynomial in size variables.

$$\mathit{SizeExpr}\ p ::= \mathcal{Q} \mid s \mid p + p \mid p - p \mid p * p \qquad s \in \mathit{SizeVar}$$

As usual $\mathcal{Q}$ denotes the set of all rational numbers. As size expressions we consider polynomials with rational coefficients that are not necessary integer. Only those of them who map non-negative integers into non-negative integers have a semantic in the type system[3]. An example of a size expression with non-integer coefficients is the polynomial for `progression` function above.

For instance, type $\texttt{[}\alpha\texttt{]}^4$ represents a list containing four elements of some type α and $\texttt{[Int]}^{(s_1-s_2)^2}$ represents a list of integers of size $(s_1-s_2)^2$ where s_1 and s_2 are size variables. Size expressions are subject to the standard associativity, commutativity

[3] In the earlier version of this paper [13] we considered only integer polynomials.

and distributivity laws for addition and multiplication. Types with negative sizes have no meaning.

We do not have partial applications and higher-order types. First-order types are functions from tuples of zero-order types to zero-order types.

$$\mathit{FTypes}\ \tau^f ::= \tau_1\ \ldots\ \tau_n \to \tau_{n+1}$$

For example, the type of `cprod`, $\texttt{[Int]}^{s_1} \times \texttt{[Int]}^{s_2} \to \texttt{[[Int]}^2\texttt{]}^{s_1 * s_2}$ is a first-order type. In well-formed first-order types, the argument types are annotated only by size variables and the result type is annotated by size expressions in these variables. Type and size variables occurring in the result type should also occur in at least one of the argument types. Thus, the type of `cprod` is a well-formed type, whereas $\texttt{[}\alpha\texttt{]}^{s_1+s_2} \to \texttt{[}\alpha\texttt{]}^{2*s_1}$ is not, because the argument is annotated by a size expression that is not a variable.

2.2 Typing system

Previously, we have developed a sound size-aware type system and a type checking procedure for a first-order functional language with call-by-value semantics [12]. The language supports lists and integers and standard constructs for pattern matching, if-then-else branching, and let-binding.

The typing rules follow the intuition on how sizes are created and changed during evaluation. The construction of a list gives a list that is one element longer than its tail. The `then` and `else` branches of the if-statement are required to yield the same size. The same holds for the `nil` and `cons` branches of pattern matching, but that rule also takes into account that the matched list is known to be empty in the `nil` branch: when matching a list of size s, if the `cons` branch has size $s * 4$, the `nil` branch can have size $0 * 4 = 0$ because, there $s = 0$.

As in [12] the formal rules are designed conventionally for ML-like syntax. Recall that an empty list `[]` is denoted by nil, a list `x:xs` is presented as $\mathsf{cons}(x, xs)$, and pattern matching and `case`-expressions both correspond to a `match`-construct. But, still, everywhere in examples we use Haskell-like syntax.

In the formal rules, a context Γ is a mapping from zero-order program variables to zero-order types, a signature Σ is a mapping from function names to first-order types, and D is a set of Diophantine equations that keeps track of which lists are empty. A typing judgment is a relation of the form $D;\ \Gamma\ \vdash_\Sigma e : \tau$ which means that if the free program variables of the expression e have the types defined by Γ, and the functions called have the types defined by Σ, and the size constraints D are satisfied, then e will be evaluated to a value of type τ, if it terminates. For example:

$$\frac{D \vdash p = p' + 1}{D;\ \Gamma,\ hd : \tau,\ tl : \texttt{[}\tau\texttt{]}^{p'}\ \vdash_\Sigma \mathsf{cons}(hd, tl) : \texttt{[}\tau\texttt{]}^{p}}\ \textsc{Cons}$$

$$\frac{\Gamma(x) = \texttt{Int} \qquad D;\ \Gamma \vdash_\Sigma e_t : \tau \qquad D;\ \Gamma \vdash_\Sigma e_f : \tau}{D;\ \Gamma \vdash_\Sigma \mathsf{if}\ x\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f : \tau}\ \textsc{If}$$

$$\frac{\begin{array}{c} p = 0,\ D;\ \Gamma,\ x : [\tau']^p \vdash_\Sigma e_{\mathsf{nil}} : \tau \\ hd, tl \notin dom(\Gamma) \qquad D;\ \Gamma, hd : \tau',\ x : [\tau']^p,\ tl : [\tau']^{p-1} \vdash_\Sigma e_{\mathsf{cons}} : \tau \end{array}}{\begin{array}{l} D; \Gamma, x : [\tau']^p \vdash_\Sigma \mathsf{match}\ x\ \mathsf{with} \mid \mathsf{nil} \Rightarrow e_{\mathsf{nil}} \quad : \tau \\ \qquad\qquad\qquad\qquad\qquad\quad \mid \mathsf{cons}(hd, tl) \Rightarrow e_{\mathsf{cons}} \end{array}}\ \textsc{Match}$$

Size-Aware type checking eventually amounts to checking entailments of the form $D \vdash p = p'$, which means that $p = p'$ is derivable from D in the axiomatics of the ring of integers. Because p and p' are known polynomials of universally quantified size variables, comparing them is straightforward. For instance, for the `cprod` function we obtain $s_1 = 0 \vdash s_1 * s_2 = 0$ (in the `nil` branch) and $\vdash s_1 * s_2 = s_2 + (s_1 - 1) * s_2$ (in the `cons` branch).

We formulated a syntactical condition sufficient to make type checking decidable for this system [12]. We allow *pattern matching and case expressions only for function parameters and variables bound to them by other pattern matchings and case expressions.* For instance, `cprod` and `sqdiff` satisfy this condition, since here only program arguments are matched. Case expressions on tails (of tails of ...) of function arguments are allowed as well:

```
f (x:xs)  =  case xs of
             [] -> ...
             (xx:xxs) -> ...
```

We prohibit constructs like

```
f x  =  case g(x) of
        [] -> ...
        (xx:xxs) -> ...
```

Note, that even with this restriction one can present all primitive recursive functions over lists. The operator of primitive recursion on lists (see, for instance, [4]) is defined as follows:

```
f ([],    ȳ)  =  g(ȳ)
f (x:xs,  ȳ)  =  h(x, xs, ȳ, f(xs,ȳ))
```

where `h`, `g` are already defined functions and $\bar{y}$ is a sequence of list or integer parameters.

2.3 Motivation of testing procedure for inference

Type inference in this type system is not straightforward. Given the degrees of polynomials, which hypothetically annotate types, a conventional type-inference procedure amounts to applying the typing rules to types with unknown size expressions. It generates a system of (non-linear) equations w.r.t. the coefficients. Such systems are, in general, hard to solve using conventional methods. Note, that we need an *exact solution within rational numbers*, otherwise the type-checker will

reject it.

Below we will consider an example for a degree two size polynomial, in which one finds the coefficients of the polynomial by solving in the end a system of 2 quadratic equations for one of these coefficients. It is clear that one may define functions that lead to more complicated, harder to solve systems of higher degrees and dimensions.

Testing is a natural way to construct a *linear system that defines the coefficients fully*. The rest of the paper shows how to infer coefficients (and, therefore, to solve corresponding systems) using an unconventional method based on a testing procedure.

Now, consider as an example of the complexity of the systems to solve, the function definition `nonlinear` with auxiliary functions:

```
copy:        [α]^s → [α]^s
copyfirst:   [α]^s1 × [α]^s2 → [α]^(s1*s2)
sqdiffaux:   [α]^s1 × [α]^s2 → [α]^(s1^2+s2^2−2*s1*s2)
```

where

```
copy []                  =   []
copy (x:xs)              =   x:  (copy xs)

copyfirst (xs, [])       =   []
copyfirst (xs, y:ys)     =   xs ++ (copyfirst(xs, ys))

sqdiffaux ([], ys)       =   copyfirst(ys, ys)
sqdiffaux (xs, [])       =   copyfirst(xs, xs)
sqdiffaux (x:xs, y:ys)   =   sqdiffaux(xs, ys)
```

The main function definition is as follows:

```
nonlinear ([], ys)       =   copyfirst      (copyfirst(ys, ys), [1,2,3,4])
nonlinear (xs, [])       =   copyfirst      (copyfirst(xs, xs), [1,2,3,4])
nonlinear (x:xs, y:ys)   =   sqdiffaux      (nonlinear(xs, y:ys)++(x:xs),
                                              nonlinear(x:xs, ys)++(y:ys))
                          ++ copyfirst      (copyfirst(x:xs, y:ys),[1,...,17] )
```

Assume that the size of an output list of `nonlinear` is calculated by a quadratic polynomial: $p(s_1, s_2) = a_{0,0} + a_{0,1}s_1 + a_{1,0}s_2 + a_{1,1}s_1s_2 + a_{0,2}s_1^2 + a_{2,0}s_2^2$. Given the annotated types for all the auxiliary functions, the coefficients a_{ij} are to be derived from the recurrence

$$\begin{aligned} p(0, s_2) &= 4s_2^2 \\ p(s_1, 0) &= 4s_1^2 \\ p(s_1, s_2) &= (p(s_1 - 1, s_2) + s_1 - (p(s_1, s_2 - 1) + s_2))^2 + 17s_1s_2 \end{aligned}$$

Substituting the initial $s_1 = 0$ and $s_2 = 0$ in the first two equations, one obtains:

$$\begin{cases} a_{0,0} + a_{1,0}s_2 + a_{2,0}s_2^2 = 4s_2^2 \\ a_{0,0} + a_{0,1}s_1 + a_{0,1}s_1^2 = 4s_1^2 \end{cases}$$

Simplifying the expression on the right hand side of the recurrent equation and applying the rule *two polynomials are equal if and only if the coefficients at corresponding degrees are equal*, one obtains the following non-linear system:

$$\begin{cases} a_{0,0} = 0, \;\; a_{1,0} = 0, \;\; a_{2,0} = 4, \;\; a_{0,1} = 0, \;\; a_{0,2} = 4 \\ a_{0,2} = (a_{1,1} - 2a_{0,2} + 1)^2 \\ a_{2,0} = (2a_{2,0} - a_{1,1} - 1)^2 \\ a_{1,1} = 2(a_{1,1} - 2a_{0,2} + 1)(2a_{2,0} - a_{1,1} - 1) + 17 \\ a_{0,1} = 2((a_{1,0} - a_{0,1}) + (a_{0,2} - a_{2,0}))(a_{1,1} - 2a_{0,2} + 1) \\ a_{1,0} = 2((a_{1,0} - a_{0,1}) + (a_{0,2} - a_{2,0}))(2a_{2,0} - a_{1,1} - 1) \\ a_{0,0} = ((a_{1,0} - a_{0,1}) + (a_{0,2} - a_{2,0}))^2 \end{cases}$$

Even after substitution of easily obtained from the initial conditions 5 coefficients into the rest of system, the system remains quadratic w.r.t. $a_{1,1}$. (Note that, the ninth and tenth equations, where $a_{1,1}$ occurs linearly, vanish due to reduction to $0 = 2 * 0 * (a_{1,1} - 3)$ and $0 = 2 * 0 * (-a_{1,1} + 3)$.) The system to solve is

$$\begin{cases} a_{1,1}^2 - 14a_{1,1} + 45 \;\; = 0 \\ 2a_{1,1}^2 - 27a_{1,1} + 81 = 0 \end{cases}$$

From this it follows that $a_{1,1} = 9$.

The testing approach presented below in this paper solves such systems as emerged in the example. It does not use the type system directly. Hypotheses for types are constructed based only on the observed behavior of the function. This avoids solving non-linear systems of equations directly. To validate the hypotheses we use the existing type checking algorithm (in practice, any type checker can be used). This ensures that, for terminating programs, type inference is complete with respect to the type checker.

3 Generating size hypotheses

This section develops a procedure (a "semi-algorithm") that uses run-time tests to automatically obtain a hypothesis for an output size polynomial, given its maximum degree. This hypothesis is correct if the output size is in fact a polynomial of the same or lower degree. In section 4, this is combined with the type checker from section 2 to obtain a size-aware type inference procedure.

The essence of the problem is giving the conditions under which a set of data points has a unique polynomial interpolation and constructing an algorithm to find points satisfying these conditions. This is not trivial since for the case of nested lists we have no information on the inner list when the outer list is empty.

3.1 Interpolating a polynomial

Looking at the sizes of the arguments and results of some tests of the `cprod` function gives the impression that the size of the outer list in the output is always the product of the sizes of the arguments. More specifically, if $p_1(s_1, s_2)$ is the size of the outer list given arguments of size s_1 and s_2, tests yielding $p_1(1,3) = 3$, $p_1(4,6) = 24$, and $p_1(3,5) = 15$ may be interpolated to $p_1(s_1, s_2) = s_1 * s_2$. Such a hypothesis can also be derived automatically by fitting a polynomial to the size data. We are looking for the polynomial that best approaches the data, i.e., the polynomial interpolation. The polynomial interpolation is unique under some conditions on the data, which are explored in polynomial interpolation theory [3,9]. If the true size expression is polynomial and the degree of the unique polynomial interpolation is high enough, the interpolating polynomial coincides with the true size expression.

We seek a condition under which the interpolation is unique. In the well-known univariate case this is simple. A polynomial $p(x)$ of degree m with coefficients $a_1, \ldots, a_{m+1}$ can be written as follows:

$$a_1 \; + \; a_2\, x \; + \; \ldots \; + \; a_{m+1}\, x^m \; = \; p(x)$$

The values of the polynomial function in any pairwise different $m+1$ points determine a system of linear equations w.r.t. the polynomial coefficients. More specifically, given the set $\big(x_i, p(x_i)\big)$ of pairs of numbers, where $1 \leq i \leq m+1$, and coefficients $a_1, \ \ldots \ , a_{m+1}$, the set of equations can be represented in the following matrix form, where only the a_i are unknown:

$$\begin{pmatrix} 1 & x_1 & \cdots & x_1^{m-1} & x_1^m \\ 1 & x_2 & \cdots & x_2^{m-1} & x_2^m \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & x_m & \cdots & x_m^{m-1} & x_m^m \\ 1 & x_{m+1} & \cdots & x_{m+1}^{m-1} & x_{m+1}^m \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \\ a_{m+1} \end{pmatrix} = \begin{pmatrix} p(x_1) \\ p(x_2) \\ \vdots \\ p(x_m) \\ p(x_{m+1}) \end{pmatrix}$$

The determinant of the left matrix, contains the measurement points, is called the Vandermonde determinant. For pairwise different points $x_1, \ldots, \ x_{m+1}$ it is non-zero. This means that, as long as the output size is measured for $m+1$ different input sizes, there exists a unique solution for the system of equations and, thus, a unique interpolating polynomial.

The conditions under which there exists a unique polynomial that interpolates *multivariate* data are not so trivial. A polynomial of degree m and dimension n (the number of variables) has $N_m^n = \binom{m+n}{n}$ coefficients. The condition under which a set of data uniquely determines a polynomial interpolation is stated as a condition on a set of *nodes* $W = \{\bar{w}_i : i = 1, \ldots, N_m^n\}$, the input sizes for which a measurement is done, such that for every set of associated measurement data $\{f_i : i = 1, \ldots, N_m^n\}$, there is a unique polynomial $p(\bar{w}) = \Sigma_{0 \leq |j| \leq m} a_j \bar{w}^j$ with total

degree m which interpolates the given data at the nodes [3]. That is, $p(\bar{w}_i) = f_i$, where $1 \leq i \leq N_m^n$. Here $\bar{w}^j = w_1^{j_1} \dots w_n^{j_n}$, $|j| = j_1 + \dots + j_n$ is the usual multivariate notation. In the next subsections, node configurations that satisfy this condition are defined, starting with bivariate polynomials and ending with the general case.

3.2 Measuring bivariate polynomials

For a two-dimensional polynomial of degree m, the condition on the nodes that guarantees a unique polynomial interpolation is as follows. In the input space, there are $m+1$ lines, each containing $m+1, \dots, 1$ of the nodes, respectively, and the nodes do not lie on the intersections of the lines. Such a configuration is depicted for parallel lines in figure 2a. This corresponds to the **NCA** configuration studied, for instance, by Chui [3].

Definition 3.1 [Two-dimensional node configuration] There exist lines in the input space, $\gamma_1, \dots, \gamma_{m+1}$, such that $m+1$ nodes of W lie on γ_{m+1}, m nodes of W lie on $\gamma_m \setminus \gamma_{m+1}$, ..., and 1 node of W lies on $\gamma_1 \setminus (\gamma_2 \cup \dots \cup \gamma_{m+1})$.

Assuming the function terminates on all inputs, such points can be found algorithmically, at least for outermost lists, using a triangle of points on parallel lines (figure 2b).

An example of the two dimensional case is the `cprod` function from the introduction. Standard type inference and annotating gives the following type:

$$\texttt{cprod} : \texttt{[}\alpha\texttt{]}^{s_1} \times \texttt{[}\alpha\texttt{]}^{s_2} \rightarrow \texttt{[[}\alpha\texttt{]}^{p_2(s_1,s_2)}\texttt{]}^{p_1(s_1,s_2)}$$

We derive that $p_1(s_1, s_2) = s_1 * s_2$ assuming p_1 is a quadratic polynomial:

$$p_1(s_1, s_2) = a_{0,0} + a_{0,1}s_1 + a_{1,0}s_2 + a_{1,1}s_1s_2 + a_{0,2}s_1^2 + a_{2,0}s_2^2$$

Running the function at the six nodes from figure 2b gives the following results:

s_1	s_2	`x`	`y`	`cprod x y`	$p_1(s_1, s_2)$	$p_2(s_1, s_2)$
0	0	`[]`	`[]`	`[]`	0	–
0	1	`[]`	`[1]`	`[]`	0	–
0	2	`[]`	`[1, 1]`	`[]`	0	–
1	0	`[1]`	`[]`	`[]`	0	–
1	1	`[1]`	`[1]`	`[[1, 1]]`	1	2
2	0	`[1,1]`	`[]`	`[]`	–	0

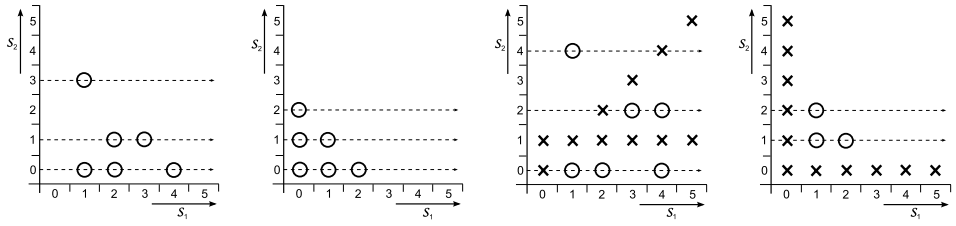

Fig. 2. (a) A node configuration that has a unique two-dimensional polynomial interpolation (b) A more systematic node configuration that has a unique two-dimensional polynomial interpolation (c) Undefined points complicate finding a node configuration (d) Undefined measurements for the pairs in the output of `cprod`.

This defines the following linear system of equations for the coefficients of p_1:

$$\begin{aligned} a_{0,0} &= 0 \\ a_{0,0} + a_{0,1} + a_{0,2} &= 0 \\ a_{0,0} + 2a_{0,1} + 4a_{0,2} &= 0 \\ a_{0,0} + a_{1,0} + a_{2,0} &= 0 \\ a_{0,0} + a_{0,1} + a_{1,0} + a_{0,2} + a_{1,1} + a_{2,0} &= 1 \\ a_{0,0} + 2a_{1,0} + 4a_{2,0} &= 0 \end{aligned}$$

The unique solution is $a_{1,1} = 1$ with the rest of the coefficients zero. Thus, we obtain the correct $p_1(s_1, s_2)$ equal to $s_1 * s_2$.

This procedure is relatively straightforward. However, there is a problem in repeating it for p_2. There are cases in which nodes have no corresponding output size (the dashes in the table). `cprod` only *partially* defines p_2, because the size of the inner lists can only be determined when there is at least one such a list. Thus, the outer list may not be empty. As can be seen in figure 2d, for `cprod` this is always the case when one of the two input lists is empty. In the next section, we show that, despite this, it is still possible to always find enough measurements and give an upper bound on the number of nodes that have to be searched.

3.3 Handling partial definedness

From the example in the previous section, it is clear that care should be taken when searching for hypotheses for output types with nested lists. In general, for $[\ldots[\alpha]^{p_k}\ldots]^{p_1}$ we will not find a value for p_j at a node if one of the outer polynomials, p_1 to p_{j-1}, is zero at that node. Thus, the nodes where p_1 to p_{j-1} are zero should be excluded from the testing process. Here, we show that, despite this, it is always possible to find enough nodes so that it becomes possible to construct an algorithm to find them.

First note that nested lists with the size of the outer list a constant zero, like $[[\tau]^{q}]^{0}$, needs special treatment. If a type-checker rejects this type with an *arbitrary*

instantiation of q, then the outer polynomial is not a constant zero. This is due to the semantics of empty lists, which is the same (an empty sequence) for any type.

Remember that we are searching parallel lines $p(x, i)$ for the node configuration. Then, for any non-zero polynomial there is a finite number of lines $y = i$, which we will call *root lines*, where $p(x, i) = 0$ (see lemma 3.2).

Lemma 3.2 *A polynomial $p(x, y)$ of degree m that is not constant 0 has at most m root lines $y = i$, such that $p(x, i) = 0$.*

Proof. Suppose there are more than m root lines. Then, it is easy to pick $1, \ldots, m+1$ nodes on $m + 1$ root lines. With these nodes, at which $p(x, y) = 0$, the system of linear equations for the coefficients of p will have the zero-solution, that is, all the coefficients of p will be zeros. This contradicts the assumption that p is not constant 0. □

Because of this property, diagonal search can always find as many nodes (x, y) as desired, such that $p(x, y) \neq 0$ (see figure 2c, where roots are marked with crosses). In fact, without requiring diagonal search, we can give a limit on the number of parallel lines $y = i$ and nodes on them that have to be searched at most. Essentially, we just try to find a triangle shape (as in figure 2b) while skipping all crosses. First, we show that for a nested list type $[[\alpha]^q]^p$ with bivariate polynomial sizes q and p, only the nodes in $[0, \ldots, m_1 + m_2] \times [0, \ldots, m_1 + m_2]$ have to be searched to determine q, where m_1 and m_2 are the degrees of p and q respectively.

Say one needs to find coefficients of an output type $[[\alpha]^q]^p$, and let $n = 2$ be the amount of variables, m_1 be the degree of $p(x, y)$ and m_2 be the degree of $q(x, y)$. One looks for test points for q that determine a unique polynomial interpolation at places where $p(x, y) \neq 0$. We restrict ourselves to lines γ parallel to the x-axis and we look for $(m_2 + 1)(m_2 + 2)/2$ data points satisfying the condition from definition 1.

Lemma 3.3 *When looking for test points for a polynomial $q(x, y)$ that determine a unique polynomial interpolation at places where another polynomial $p(x, y) \neq 0$, it is sufficient to search the lines $y = 0, \ldots, y = m_1 + m_2$ in the square $[0, \ldots, m_1 + m_2] \times [0, \ldots, m_1 + m_2]$.*

Proof. For the configuration it is sufficient to have m_2+1 lines with at least m_2+1 points where $p(x, y) \neq 0$. Due to lemma 3.2 there are at most m_1 lines $y = i$ such that $p(x, i) = 0$, so at least $m_2 + 1$ are not root lines for p. The polynomial $p(x, j)$, with $y = j$ not a root line, has at most degree m_1, thus $y = j$ contains at most m_1 nodes (x, j), such that $p(x, j) = 0$. Otherwise, it would have been constant zero, and thus a root line. Hence, this leaves at least $m_2 + 1$ points on these lines for which p is not zero. □

This straightforwardly generalizes to all nested types with polynomials in two variables, say $[\ldots[\alpha]^{p_k}\ldots]^{p_1}$. If we want to derive the coefficients of p_i, searching the square of input values $[0, \ldots, \Sigma_{i=1}^{k} m_i] \times [0, \ldots, \Sigma_{i=1}^{k} m_i]$ suffices, where m_i is the degree of p_i. Each p_j has at most m_j root lines, so there are at most $\Sigma_{j=1}^{i-1} m_j$ root

lines for $p_1, \ldots, p_{i-1}$. Also, each of the p_j can have at most m_j zeros on a non root line. Hence, since the length of the search interval is $\Sigma_{j=1}^{k} m_j + 1$, there are always $m_i + 1$ values known.

For `cprod` there are two size expressions to derive, p_1 for the outer list and p_2 for the inner lists. Deriving that $p_1(s_1, s_2) = s_1 * s_2$ is no problem. Because p_1 has roots for $s_1 = 0$ and for $s_2 = 0$, these nodes should be skipped when measuring p_2 (see figure 2d).

3.4 Generalizing to n-dimensional polynomials

The generalization of the condition on nodes for a unique polynomial interpolation to polynomials in n variables, is a straightforward inductive generalization of the two-dimensional case. In a hyperspace there have to be hyperplanes, on each of which nodes lie that satisfy the condition in the $n-1$ dimensional case. A hyperplane K_j^n may be viewed as a set in which test points for a polynomial of $n-1$ variable of the degree j lie. There must be $N_j^{n-1} = N_j^n - N_{j-1}^n$ such points. The condition on the nodes is defined by:

Definition 3.4 [n-dimensional node configuration] The **NCA** configuration for n variables (n-dimensional space) is defined inductively on n [3]. Let $\{x_1, \ldots, x_{N_m^n}\}$ be a set of distinct points in $\mathcal{R}^n$ such that there exist $m+1$ hyperplanes K_j^n, $0 \leq j \leq m$ with

$$x_{N_{m-1}^n+1}, \ldots, x_{N_m^n} \in K_m^n$$

$$x_{N_{j-1}^n+1}, \ldots, x_{N_j^n} \in K_j^n \setminus \{K_{j+1}^n \cup \ldots \cup K_m^n\}, \text{for } 0 \leq j \leq m-1$$

and each of set of points $x_{N_{j-1}^n+1}, \ldots, x_{N_j^n}$, $0 \leq j \leq n$, considered as points in $\mathcal{R}^{n-1}$ satisfies **NCA** in $\mathcal{R}^{n-1}$.

Thus, similarly to lines in a square in the two dimensional case, parallel hyperplanes in a hyperspace have to be searched. Using a reasoning similar to the two-dimensional case one can show that it is always sufficient to search a hypercube with sides $[0, \ldots, \Sigma_{i=1}^{k} m_i]$. The proof is also straightforwardly generalized.

4 Automatically inferring size-aware types

The type checking procedure from section 2 and the size hypothesis generation from section 3 are combined into an inference procedure for an increasing degree of polynomials. The procedure is semi-decidable: it only terminates when the function is well-typable in the type system of the type checker used. In this sense the procedure is complete w.r.t. a type-checker: if a function definition is well-typed, the type will be found.

Recently, we have developed a demonstrator for the inference procedure. The demonstrator (including its source code and the corresponding Java docs) is accessible on `www.aha.cs.ru.nl`.

4.1 The procedure

For any shapely program, the underlying type (the type without size annotations) can be derived by a standard type inference algorithm [10]. After straightforwardly annotating input sizes with size variables and output sizes with size expression variables, we have for example

$$\texttt{cprod} : [\alpha]^{s_1} \times [\alpha]^{s_2} \to [[\alpha]^{p_2(s_1,s_2)}]^{p_1(s_1,s_2)}$$

To derive the size expressions on the right hand side we use the following procedure. First, the maximum degree of the occurring size expressions is assumed, starting with zero. Then, a hypothesis is generated for each size expression. This is done from the "outside in" on the annotations for output types, because of the problems with partially definedness described in section 3.3. After hypotheses have been obtained for all size expressions they are added to the type and this hypothesis type is checked using the type checking algorithm. If it is accepted, the type is returned. If not, the procedure is repeated for a higher degree.

Figure 3 shows the procedure in pseudo-code. The *TryIncreasingDegrees* function generates *GetSizeAwareType* and checks *CheckSizeAwareType* hypotheses of increasing degrees. A size expression is derived by selecting a node configuration *GetNodeConf*, running the tests for these nodes *RunTests*, and deriving the size polynomial from the test results *DerivePolynomial*.

Note that if the assumed degree is lower than the true degree, then the derived polynomials may be wrong. It will be later rejected by a type checker, or the nodes where the size annotations are fully defined cannot be determined correctly. It may happen that the node configuration has "too many" points where the size expression is undefined so the test results do not provide enough information to uniquely infer the inner polynomial(s). In that case one increases the degree and continue inference.

If a type is rejected, this can mean two things. First, the assumed degree was too low and one of the size expressions has a higher degree. That is why the procedure continues for a higher degree. Another possibility is that one of the size expressions is not a polynomial (the function definition is not shapely) or that the type cannot be checked due to incompleteness. In that case the procedure will not terminate. Fortunately, in practice a suitable stopping criterion may be known. If the function is well-typable, the procedure will eventually find the correct size-aware type and terminate.

4.2 Examples

Based on the results of Section 3 we define a *theoretical maximum* on the number of tests in the following way. Let $m = max\{m_1, \ldots, m_k\}$. The theoretical maximum is the upper bound $(1 + km)^n$. In practice, as a rule, the number of tests needed to define all the output polynomials for a given function is significantly closer to the theoretical lower bound defined by $\binom{m+n}{n}$ which is the number of coefficients of the polynomial with n variables and degree m.

Function: TryIncreasingDegrees

Input: a degree m, a function definition `f`

Output: the size-aware type of that function

TryIncreasingDegrees(m, f) =

let *type* = InferUnderlyingType(f)
atype = AnnotateWithSizeVariables(*type*)
vs = GetOutputSizeVariables(*atype*)
stype = GetSizeAwareType(m, f, *atype*, *vs*, [ ])
in if (CheckSizeAwareType(*stype*, f)) then *stype*
else TryIncreasingDegrees(m+1, f)

Function: GetSizeAwareType
Input: a degree, m the function definition `f` with its annotated type, a list of unknown size annotations, and the polynomials already derived
Output: the size-aware type of that function if the degree is high enough

GetSizeAwareType(m, f, *atype*, [ ], *ps*) =

AnnotateWithSizeExpressions(*atype*, *ps*) // The End

GetSizeAwareType(m, f, *atype*, v:*vs*, *ps*) =

let *nodes* = GetNodeConf(m, *atype*, *ps*)
results = RunTests(f, *nodes*)
p = DerivePolynomial(m, v, *atype*, *results*)
in GetSizeAwareType(m, f, *atype*, *vs*, p:*ps*)

Fig. 3. The type inference procedure in pseudo-code

The minimum number of tests will be enough when there are no empty nested lists and the set of test data is *well-formed* i.e. corresponds to an **NCA** configuration. In the tests for the `append` example this was e.g. the case. The type of `append` does not contain nested lists.

In practice, the number of tests is usually closer to the minimum than to the maximum since the case of nested non-empty lists occurs more frequently than the empty case. Moreover, one may store and reuse the results of testing. So, only a few extra tests are needed.

The algorithm is illustrated by the following functions: `++` (append), `progression`, `cprod`, `sqdiff` and `competition`. The function `competition` generates a competi-

function	m	nr. of tests	type suggested	type checker
`append`	0	1 (1)	$[\alpha]^{s_1} \times [\alpha]^{s_2} \to [\alpha]^0$	reject
(n = 2, k = 1)	1	3 (4)	$[\alpha]^{s_1} \times [\alpha]^{s_2} \to [\alpha]^{s_1+s_2}$	accept
`progression`	0	1 (1)	$[\alpha]^{s} \to [\alpha]^0$	reject
(n = 1, k = 1)	1	2 (2)	$[\alpha]^{s} \to [\alpha]^{s}$	reject
	2	3 (3)	$[\alpha]^{s} \to [\alpha]^{\frac{s*(1+s)}{2}}$	accept
`cprod`	0	1 (1)	$[\alpha]^{s_1} \times [\alpha]^{s_2} \to [[\alpha]^-]^0$	reject
(n = 2, k = 2)	1	3 (9)	$[\alpha]^{s_1} \times [\alpha]^{s_2} \to [[\alpha]^-]^0$	reject
	2	11 (25)	$[\alpha]^{s_1} \times [\alpha]^{s_2} \to [[\alpha]^2]^{s_1*s_2}$	accept
`sqdiff`	0	1 (1)	$[\alpha]^{s_1} \times [\alpha]^{s_2} \to [[\alpha]^-]^0$	reject
(n = 2, k = 2)	1	4 (9)	$[\alpha]^{s_1} \times [\alpha]^{s_2} \to [[\alpha]^2]^0$	reject
	2	8 (25)	$[\alpha]^{s_1} \times [\alpha]^{s_2} \to [[\alpha]^2]^{(s_1-s_2)^2}$	accept
`competition`	0	1 (1)	$[\alpha]^{s} \to [[\alpha]^-]^0$	reject
(n = 1, k = 2)	1	2 (3)	$[\alpha]^{s} \to [[\alpha]^-]^0$	reject
	2	5 (5)	$[\alpha]^{s} \to [[\alpha]^2]^{\frac{s^2-s}{2}}$	accept

Table 1
Type construction for four functions (n is the number of input variables, k the number of output polynomials). Both the actual number of *test runs* and the theoretical maximum (between parentheses) $(1 + km)^n$ are given.

tion in which every team plays a home and away match against every other team:

```
competition  xs          =  randomize_order (competition' xs [])
competition' ([], ys)    =  []
competition' (x:xs, ys)  =  pairs (x, xs) ++ competition' (xs, x:ys)
```

Table 1 gives for each function all hypotheses generated for each needed iteration.

5 Discussion and Future Work

The procedure currently has three apparent limitations. First, the procedure has two possible sources of non-termination. Second, it only works for exact sizes and

not for upper bounds. Third, it is developed for a first-order functional language with lists as the only supported data structures. Here, these issues are discussed and improvements are suggested.

5.1 Sources of Nontermination

Because the procedure uses run-time tests, it does not terminate when one of these tests does not terminate. In practice, however, this is not an important problem, because the analysis will typically be run on a stable product where non-termination should be rare. Just in case, a termination analysis can be done first or the procedure may be adapted to start looking for replacement tests if evaluation of a test takes too long and non-termination is suspected. In general, this problem is very related to the active research field of test-case construction.

The second source of nontermination is the iteration over increasing degrees of polynomials. If none of the generated types is accepted by the checker, either because the function definition is not shapely or due to incompleteness, the procedure in principle does not stop. In practice, often an upper bound can be put on the degree because only size expressions of low degree are desired.

We do not expect that the procedure may be easily adopted for *lazy languages*. All functions must be well-typed. We must be able to infer types for each of them. If one of the functions does not terminate during *its* inference-via-testing then the procedure for the main function gets stuck. For a strict language this coincides with an incorrect, non-terminating program. For a lazy language the main function will not necessarily be non-terminating. One of the possible ways to continue testing in such cases is to involve *dependent typing*.

5.2 Scaling up to real programming

5.2.1 Non-shapely programs

The current hypothesis generation procedure relies on the limitation to shapely programs; output sizes need to be exactly polynomial in the input size. In practice many programs are not shapely, but still have a polynomial upper bound. Consider inserting an element in a set. This increases the set size by one only if the element was not in it. Its actual upper bound is:

$$\texttt{insert} : [\alpha]^s \times \alpha \to [\alpha]^{s+1}$$

To extend our approach to such upper bounds, we are studying program transformations that transform an unshapely function into a shapely function with the strict size dependency corresponding to an upper bound of the size dependency of the original function. For instance, the `insert` function would be transformed into a shapely function that always inserts the element. We believe that in many practical cases the testing approach combined with program transformations will succeed in providing good upper bounds.

5.2.2 General data structures

In this paper, we presented the procedure for a simple functional language over lists. We plan to extend and implement the procedure for an existing language with more general data structures. Good candidates are XML transformation languages [16,7] because such transformations are very likely to be shapely. For these applications, the general type inference procedure will stay the same. The only requirement is that a type checker for such a language is developed.

5.2.3 Higher-order functions

A natural extension of our first-order analysis is to allow higher-order functions. A consequence for the analysis could be the inclusion of operators in a space of rational polynomials. Solving decidability issues for such an extension seems to be rather complicated.

Another option might be to consider program transformations that translate higher-order programs to equivalent first-order programs. This could break modularity but it might be a practical solution for "full" programs which take data input and produce data output.

6 Conclusion

We have developed a procedure (and a demonstrator for it) to infer static non-monotone size-aware types through interpolating data from run-time tests. The dynamically generated types are only accepted after checking them by a formal type checking algorithm. So, the types are static: the size expressions hold for every possible future run of the program.

Our key idea was the use of a dynamic testing procedure to generate hypotheses for the size-aware types. This replaces an otherwise infeasible to define formal type inference procedure and essentially reduces type inference to type checking. As a consequence, type inference is complete with respect to type checking.

6.1 Related work

Some interesting initial work on inferring size relations within the output of XML transformations has been done by Su and Wassermann [14]. Although this work does not yield output-on-input dependencies, it is able to infer size relations within the output type, for instance if two branches have the same number of elements. Herrmann and Lengauer presented a size analysis for functional programs over nested lists [8]. However, they do not solve recurrence equations in their size expressions, as this is not important for their goal of program parallelization.

Other work on size analysis has been restricted to monotone dependencies. Research by Pareto has yielded an algorithm to automatically check linear sized types where size expression are upper bounds [11]. Inspired by this work Chin and Khoo [2] devised a fixed-point method for inferring linear lower and upper bounds. Construction of non-linear upper bounds using a traditional type system approach has

been presented by Hammond and Vasconcellos [15], but this work leaves recurrence equations unsolved and is limited to monotone dependencies. Debray and Lin [5] reduce inference of monotone size-relations for logical programs to solving difference equations and rely on external solving as well. We will study their generic size functions in order to extend our method. The work on quasi-interpretations by Amadio [1] also requires monotone dependencies.

Acknowledgement

The authors would like to thank the students of Radboud University, Nijmegen – Willem Peters, Bob Klaase, Elroy Jumpertz – without whom implementation of the on-line demonstrator would have been impossible.

References

[1] Amadio, R.: "Max-plus quasi-interpretations"; Typed Lambda Calculi and Applications, Lecture Notes in Computer Science 2701 (2003), 31–45.

[2] Chin, Wei-Ngan Chin and Khoo, Siau-Cheng: "Calculating Sized Types"; Higher-Order and Symbolic Computation, 14(2-3)(2001), 261–300.

[3] Chui, C. and Lai, H. C.: "Vandermonde determinant and Lagrange interpolation in R^s"; Nonlinear and convex analysis (1987), 23–35.

[4] Colson L.: "On List Primitive Recursion and the Complexity of Computing"; BIT Numerical Mathematics, 32 (1) (1992), 5–9.

[5] Debray, Saumya K. and Lin, Nai-Wei: "Cost analysis of logic programs"; ACM Transactions on Programming Languages and Systems, 15(5)(1993), 826–875.

[6] van Eekelen, M., Shkaravska, O. and van Kesteren, R. and Jacobs, B and Poll, E. and Smetsers, S.: "Aha: Amortized heap space usage analysis"; Proceedings of the Eighth Symposium on Trends in Functional Programming, TFP 2007, XVI–1–16. Submitted for selected papers.

[7] Frisch, A.: "Ocaml + XDuce"; Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming (2006), 192–200.

[8] Herrmann, C. A. and Lengauer, C.: "A transformational approach which combines size inference and program optimization"; Semantics, Applications, and Implementation of Program Generation (SAIG'01), Lecture Notes in Computer Science 2196(2001), 199–218.

[9] Lorenz, R. A.: "Multivariate Birkhoff Interpolation"; Lecture Notes in Math. 1516(1992)

[10] Milner, R.: "A theory of type polymorphism in programming"; Journal of Computer and System Sciences, 17(3)(1978), 348–375.

[11] Pareto, R.: "Sized Types. Dissertation for the Licentiate Degree in Computing Science"; Chalmers University of Technology, Göteborg (1998)

[12] Shkaravska, O., van Kesteren, R. and van Eekelen, M.: "Polynomial size analysis of first-order functions"; Proceedings of International Conference on Typed Lambda Calculi and Applications (2007); Lecture Notes in Computer Science 4583 (2007), 351-366

[13] van Kesteren, R., Shkaravska, O. and van Eekelen, M.: "Inferring static non-monotonically sized type through testing"; 16^{th} International Workshop on Functional and (Constraint) Logic Programming (WFLP07), Paris, France, ed. Rachid Echahed; CNAM, France, 123-139.

[14] Su, Z. and Wassermann, C.: "Type-based inference of size relationships for xml transformations"; Technical Report U CD//CSE-2004-8, UC Davis (2004)

[15] Vasconcelos, P. B. and Hammond, K.: "Inferring cost equations for recursive, polymorphic and higher-order functional programs"; Lecture Notes in Computer Science 3145(2004), 86–101.

[16] Wadler Ph.: "A formal semantics of patterns in XSLT and XPath"; Markup Lang., 2(2)(2000), 183–202.